Dpx

PCT AS FILED
TRUS3001

10/540501
JC17 Rec'd PCT/PTO 23 JUN 2005

METHOD FOR SECURING COMPUTER SYSTEMS INCORPORATING A CODE INTERPRETATION MODULE

The present invention relates to securing computer systems comprising at least one code interpretation module and storage memory capacities for the code to be interpreted.

More specifically, its object is to solve problems for securing computer systems comprising at least one code interpretation module (a code being defined as a structured set of instructions) which will simply be called "interpreter" in the following (hardware interpreter: microcontroller, microprocessor or software: virtual machine) and storage memory capacities for the code to be interpreted (or interpreted code).

Said code may be directly written by a programmer, may be obtained automatically (which will be called code generation) from a source code in a language which is generally of a higher level or it may even result from a combination of automatic production and manual interventions.

Generally, it is known that most of the inventorized attacks against such computer systems are based on physical measurements (electromagnetic emission, etc.) during execution and require synchronization with the interpreted code. In other words, the intruder must determine at what time the interpreter is found executing certain functionalities of the code. Among the most known techniques, those developed in order to find a key in cryptographic algorithms by passively spying the physical emission of a circuit may be cited: in particular the attacks of the simple power analysis (SPA) and differential power analysis (DPA) type have been successfully used for discovering DES ("Data Encryption Standard") keys. As an example, on an embedded Java platform ("Java Card", "JEFF", "J2ME" ...) these attacks may be used in order to try to obtain information on secrets handled by the virtual Java machine. These secrets may concern both the confidential data and the Java code itself.

More specifically, the object of the invention is therefore to suppress these drawbacks.

For this purpose, it proposes making attacks based on physical measurements and/or requiring synchronization with the interpreted code, more difficult, by introducing alternatives in the execution times of the interpreted code and the measurable physical imprints (for example, and not exclusively, electromagnetic emission, etc.).

According to the invention, this method essentially involves two types of alternatives in the execution times of the interpreted codes, in the following way:

- by causing at certain places of an interpreted code bypasses towards new portions of code (which do not belong to the original code) in order to complicate the synchronization and the physical imprint of the execution, or
- by proposing a plurality of implementations of certain instructions, each requiring a different execution time and/or having a different physical imprint and providing an identical result, so that two executions of this instruction within a same code may be performed by two different implementations.

Hence, by introducing distortions in the execution times and by modifying the physical effect of the execution, both types of the above alternatives will make any correlation attempt more difficult between the observed physical expressions of an interpreted code and its functionalities.

Advantageously, this method will be able to make the apparently executed code, different at each execution, and will therefore make the discovery of the actual code of the application, more difficult.

This method may involve:

- for the first alternative:
  - two modes for introducing "bypass codes",
  - four modes for realizing "bypass codes",
- for the second alternative:
  - two modes for introducing "multiple implementations" of certain instructions,
  - three modes for realizing "alternative codes" with a variable physical imprint and duration.

As regards the first alternative, the first mode for introducing "bypass codes" consists of introducing one or more specific so-called "bypass" instructions in certain particular locations of the code. This introduction may be made either manually or automatically upon generating the code. In the latter case, the code generator may be guided in order to produce these instructions by annotations inserted by the programmer in the source code and allowing the designation of portions of sensitive code (for example, and in a non-limiting way, encryption or access rights checking procedures). Execution of a bypass instruction by the interpreter causes branching towards an associated bypass code. This first method may also be improved by attaching different levels of security to bypass instructions and by associating them with all the more complex (or defensive with regards to security attacks as described above) bypass codes since their security level is high.

Concerning the first alternative, the second mode for introducing "bypass codes" consists of introducing the bypass code in the implementation of the interpreter itself: between the executions of two consecutive instructions of the code, the interpreter executes the bypass code, either systematically or selectively or randomly. For example, it may execute this code only when certain sensitive methods are called (typically from so-called API (application program interface) libraries).

The advantage of the first mode is to allow selective introduction of the executions of bypass code which leads to less penalty in terms of execution times if the number of such bypasses is small. It also allows implementation of so-called "discretionary" security policies, i.e., at the discretion of the applications.

On the other hand, the second mode will be more advantageous when the number of desired bypasses is large because the implementation of the method in the interpreter itself may then be optimized. Moreover, it allows implementation of so-called "proxy" security policies where checks are uniformly imposed on all the applications.

Both previous aforesaid introduction modes require the introduction of a bypass code. The invention proposes four modes to realize these bypass codes so that they introduce alternatives in the execution times and the measurable physical imprints.

Concerning the first alternative, the first mode for realizing "bypass codes" with physical imprint and variable duration consists of performing a so-called "superfluous" calculation depending on data known at execution (which may therefore differ at each execution). The superfluous calculation should be without any effect on the final result of the execution of the interpreter. A simple example of such a calculation is a parity test for a dynamic datum (known at execution) which may either lead to a void action, or to the adding of an item to a stack followed by its immediate removal. It should be noted that the number of possible actions is not necessarily limited to two. A large possible number of actions will lead to significant variability in the execution time and the physical imprint of the bypass code.

The second mode for realizing "bypass codes" improves the first mode by providing it with a random draw of an extra datum during the execution of the superfluous calculation, said extra datum being used in the calculation performed by the bypass code (for example in a test of said code). This random draw has a new variable item and makes the execution time and the physical imprint of the bypass code, even less predictable.

The third mode for realizing “bypass codes” improves the efficiency of the two preceding ones by replacing the test for deciding on the next action with a branching in a so-called indirection table, i.e., containing the addresses of possible actions, at an index calculated from variable items (dynamic datum and/or result from a random draw).

The fourth mode for realizing “bypass codes” improves the first embodiment (and therefore the three other ones) by considering a superfluous calculation which, while remaining without any effect on the final result, has external characteristics (physical imprint) of a particular sensitive calculation (for example encryption or decryption) without any relationship with the actual code of the application. Such a superfluous calculation enables an attacker to be fooled, who would attempt to infer secrets by measuring the physical effect of the execution of the application. Such a method may be described as a “software decoy” since its goal is to induce in error the attackers by making them believe in the presence of said sensitive calculation in the actual code of the application. This mode may simply be achieved by implementing the relevant sensitive calculation without retaining its result.

Concerning the second alternative, the first mode for “introducing multiple implementations” of certain instructions, consists of enriching the set of instructions recognized by the interpreter with a plurality of implementations for a given instruction. These implementations will be achieved so as to have different physical imprints and different execution times while producing an identical result. Any of these implementations may be used in the code indiscriminately. This use may be performed either manually by programming or automatically during code generation. In the latter case, the code generator may be guided, in order to produce these instructions, by annotations inserted by the programmer into the source code and allowing designation of sensitive code portions (for example, and in a non-limiting way, encryption or access rights checking procedures). This first mode may also be improved by attaching different security levels to the implementations of instructions and associating them with all the more complex (or defensive with regard to security attacks) implementations since their security level is high.

Concerning the second alternative, the second mode for introducing "multiple implementations" of certain instructions, consists of comprising in the actual implementation of the instruction, a branching to an alternative code portion which will dynamically determine the implementation to be executed.

The advantage of the first mode is to minimize additional costs in terms of execution times as the selection of the instruction implementation to be applied is determined before execution. It also allows implementation of so-called "discretionary" security policies, i.e., at the discretion of the applications.

The advantage of the second mode is to further complicate the attacks requiring synchronization with the code since two consecutive executions of the same instruction (at the same location in the code) will be able to take different execution times and to provide different physical imprints. Moreover, this second mode allows implementation of so-called "proxy" security policies where the checks are uniformly imposed to all the applications.

Both modes do not mutually exclude each other: a realization may comprise a multiplicity of implementations for a given instruction, certain of them (or all of them) being implemented by branching to an alternative code portion dynamically determining the implementation to be executed.

The aforesaid second mode of the second alternative requires the introduction of an alternative code associated with an instruction. The invention proposes three modes for realizing this alternative code so that it introduces different implementations in the execution times and the measured physical imprint.

Concerning the second alternative, the first mode for realizing "alternative codes" with a physical imprint and variable duration consists of proposing a plurality of different implementations of the instruction and to condition the choice of the executed version to a dynamical test, i.e., depending on data known at execution. A simple example of such a calculation is a parity test of a dynamical datum (known at execution). A large number of implementations will lead to significant variability in execution time and in the physical imprint of the alternative code.

The second mode for realizing "alternative codes" improves the first mode by providing it with a random draw of a datum which is then used for achieving the test leading to the dynamical choice of the executed version. This random draw adds a new variable item and makes the execution time and the physical imprint of the alternative code, even less predictable.

The third mode for realizing "alternative codes" improves the efficiency of the two preceding ones by replacing the test for deciding on the selected version with a branching in a indirection table (containing the addresses of the available versions) at an index calculated from variable items (dynamical datum and/or result from a random draw).

Thus, by introducing alternatives in the execution times of the interpreted codes and therefore in the physical imprints, it is possible to make the attacks based on said physical imprints, more difficult so that an action coded in the implementation of the application may have different electronic signatures and occurring at variable execution times.

The implementation of the aforesaid interpreted codes will be performed on modules for interpreting software code such as virtual machines of the JAVA family and on modules for interpreting physical code of the microcontroller or microprocessor type.